Towards a Formal Approach for the Regeneration of PILOT Control System

Laurent Tchamnda Nana, Valérie-Anne Nicolas, Lionel Marcé

Towards a Formal Approach for the Regeneration of PILOT Control System

Laurent NANA TCHAMNDA, Valérie−Anne NICOLAS and Lionel MARCE
Computer Science Department, University of Brest,
20 Avenue Le Gorgeu BP 809
29285 Brest Cedex, France

ABSTRACT

PILOT (Programming and Interpreted Language Of actions for Telerobotics) is a high level language dedicated to the remote control of systems. Our team has built a complete control system for PILOT, which comprises six main modules: a human−machine interface, an interpreter, a rules generator, an evaluator, an execution module and a communication server. In this paper, we focus on the interpreter which is one of the most important parts of the system. For the initial release of the control system software, the main goal was to have a working environment in order to validate the concepts of the language PILOT. The various modules of the control system have been modeled by finite state automata and the code has been written manually. Although the experimentation carried out with the first release of the control system highlighted the benefits of PILOT for the control of mobile robots such as the robot MARC'H built by our team, some malfunctions were observed in the software and particularly in the interpreter. This paper aims at presenting the work performed, essentially based on testing, in order to detect and to correct software errors into the interpreter, at both conceptual and implementation levels. It ends by our ongoing work related to the use of Petri nets for modeling and testing the interpreter algorithms, the final goal being the generation of safe programs from "validated" models.

Keywords: Control systems, software testing, telerobotics, modeling, Petri nets.

1. INTRODUCTION

Safety is a critical issue in robots programming and is to be taken into account when designing a language for telerobotics or building software dedicated to the control of robots. In this perspective, an operational semantics of PILOT has been defined [2,3] and this makes it possible to apply formal verification techniques to PILOT programs. PILOT also integrates several features for safe programming of robots such as actions preconditions and supervising rules [6]. In the same line of thought, PILOT control system software has been designed following some software engineering rules needed to produce safe programs. In particular, the various modules of the software have been modeled by finite state automata [4]. The experimentation carried out with the first release of the control system highlighted the benefits of PILOT for the control of mobile robots [5]: easiness of use (partially due to its graphical nature), portability, possibility to deal with unexpected situations during plan execution (fault tolerance), etc.. Nevertheless, some malfunctions were observed in the software and particularly in the interpreter. In fact, for the initial software release, the main goal was to have a working environment in order to validate the concepts of the language PILOT. So, some safety related issues such as software testing were not considered. Since the initial release, several works have been performed in order to increase the safety of PILOT applications [6]. In this paper, we focus on our recent work related to the testing of PILOT interpreter's software in order to detect and to correct errors at both conceptual and implementation levels.

This paper starts by an overview of PILOT and of its control system. Then, the structure and the implementation of the interpreter are briefly described. Thereafter, our approach for detecting and correcting the interpreter software errors using static and dynamic testing is presented. The paper ends by our ongoing work related to the use of Petri nets for modeling and testing the interpreter algorithms.

2. PILOT: A LANGUAGE AND A CONTROL SYSTEM

The Language PILOT

The language PILOT is based on the notion of action. An action encapsulates an order executable by the robot, a precondition and one or more supervising rules to which processings are associated. Two kinds of actions are discriminated: elementary and continuous actions. Unlike a continuous action whose end is triggered by an enclosing primitive of the language, an elementary action generally ends when its predefined goal is reached. Whatever its kind may be, an action is only executed when its precondition rule is true (unless the operator decides to force the execution). In the same way, if during the execution of an action, one of its supervising rules becomes true, then the corresponding processing is performed (the default processing associated with a supervising rule is the stopping of the corresponding action). In practice, preconditions and supervising rules are conditions on sensors values.

The language PILOT provides control structures for plan building: sequence, conditional, iteration, parallelism and preemption. Detailed information on the language PILOT can be found in [3].

The Control System of PILOT

The control system of PILOT comprises six concurrent modules:

- The graphical interface also called man−machine interface,
- The interpreter,
- The communication server,
- The rules generator,
- The evaluator and
- The execution module or driver.

These processes communicate through sockets and shared memory and can execute either on a single computer or on a network of computers.

The man−machine interface provides different features for designing plans. It stores the plan into a memory space shared with the interpreter. The interpreter reads the plan from the shared memory and sends orders (precondition request, order to start an action, ...) to the other modules in order to achieve the plan execution. The communication server handles inter−process communications. The purpose of the rules generator is to transform character strings of precondition and supervising rules into binary trees. It stores the result into shared memory for future use by the evaluator. The rules evaluator is in charge of calculating the Boolean expressions of precondition and supervising rules. The execution module is the interface between the robot and the control system. It translates high level orders of the plan into low−level orders which are understandable by the teleoperated machine.

3. THE INTERPRETER: INITIAL STRUCTURE AND IMPLEMENTATION

The interpreter is one of the most important parts of the control system. It reads the plan stored into shared memory by the graphical interface, then it performs some requests to the other modules of the control system in order to achieve the plan execution. The behavior of the interpreter can be summarized as follows:

```
Begin interpreter
    Set interrupt handling routines
    Create structures for plan execution
    Initialize communication medium
    Loop
        MessageHandling
    End loop
End interpreter
```

The interrupt handling routines are the following:

- *Handler*: this procedure is called when signal QUIT is received. Its effect is to close the communication medium, to finalize the execution of the interpreter (closing open files, releasing allocated spaces, etc.) and to end the execution of the interpreter.
- *MessageHandling*: this procedure is called when signal SIGUSR1 is received. This signal may arrive at any time. The behavior of the procedure *MessageHandling* is the following:

```
Begin MessageHandling
    Reset interrupt handling routines
    Wait Until message received
    Case
        Message Kind = "START_EXECUTION" =>
            {Message from the graphical interface to ask to
            start the interpretation of the plan}
            Courant ← Head (Plan)
            Interpret (Plan)
        Message Kind = "PRECONDITION_ACCEPTED" =>
            {Message from the evaluator indicating that the
            precondition of an action is true}
            Send a message "START_ACTION" to request the
            launching of the related action
            Mark the action into shared memory as started
            {the graphical interface will update the graphics
            accordingly for supervision}
        Message Kind = "STOP_ACTION" =>
            {Message from the evaluator when an action is
            terminated}
            Stop the given action and interpret the primitive
            following the action terminated
            Suppress marking on the action into shared
            memory
    End case
End MessageHandling
```

In this description, we have only mentioned the messages which are useful for the understanding of the paper. The procedure *Interpret* has the following shape:

```
Begin Interpret
    Initialize various structures for actions execution and
termination handling
    While (not EndOfPlan) loop
        Process (Courant)
        While (AlreadyProcessed (Courant)) loop
            Courant ← Next (Courant)
            EndOfPlan ← IsNul (Courant)
            Exit when EndOfPlan
        End loop
    End loop
    ...
End Interpret
```

As mentioned above, the main goal of the initial release of the control system of PILOT was to validate the concepts of the language PILOT. So, very few importance was attached to the testing of the control system software. Before describing our

approach for detecting and correcting interpreter errors, we first present, in the next section, a brief overview of software testing.

4. AN OVERVIEW OF SOFTWARE TESTING

Software testing [1,7,8] is an important step in the software development process. It is principally interested in the final product of the programmer's activity, which is the program code, and to its behavior. Testing is one of the traditional means used to reach software safety. Its goal is to minimize failures appearance chances in the course of software use. Testing activity consists in:

- Either searching *statically* for simple and frequent errors: this approach is also called *software control*,
- Either defining input data which will be given to the software during its execution. These input data are also called *test data*. The set of test data generated for testing is designated as a *test data set.* The last step of testing is to compare the output data obtained to the expected output for the given software.

In order to increase the efficiency of the testing process, test data should constitute a representative sample of all the possible input data which can be submitted to the software.

The Causes of Software Errors

The software development process has an impact on the potential number and on the potential kind of errors. For example, a software built following software engineering principles and submitted to formal verifications will potentially have less errors than a software built with a less rigorous method or without formal verifications. The knowledge of the programming language used to write the programs as well as the experience of the programmers also have an impact on the errors (nature, number): an inexperienced programmer is potentially subject to more errors than an experienced programmer. Some errors are inherent in information distortion or loss throughout the development process. Failures may be due to a wrong specification, a misunderstanding of the specification, an imperfect knowledge of the programming language, ...

Classification of Errors

The set of failures which can affect software is infinite and it is difficult to classify them. Nevertheless, six classes of software errors can be defined:

- Calculation errors: for example writing "x:= x + 2" instead of "x:= y +2"
- Logic errors: wrong expression of a predicate. For example writing "if (a < b) then" instead of "if (a > b) then"
- I/O errors: wrong formatting, bad access to communication medium, etc.
- Interface errors: wrong communication between the software components (for example call to a function F1 instead of F2, wrong parameter passing, etc.).
- Data processing errors: wrong data access or wrong data handling (misuse of pointers, undefined variables, array index out of range, etc.).
- Data definition errors: wrong type in data declaration (for example, data is declared as integer whereas it should have been declared as real), error in the precision (for example, a value is defined with a simple precision instead of a double precision).

Classification of Testing Techniques

Different testing techniques exist. They can be classified according to the criteria they use to choose representative test data. Two categories are distinguished in this case:

- *Functional* techniques also called *black box* techniques: in such techniques, the production of test cases is based on the software specification, without worrying about the internal structure of the software.
- *Structural* techniques also called *white box* techniques: test cases are produced by analyzing the source code.

Testing techniques can also be classified according to the execution or not of the binary code. The following categories are distinguished:

- *Dynamic testing* techniques: the binary code is executed and the real behavior of the program is examined.
- *Static testing* techniques: the passive form of the program (source code) is examined.

Testing process often use a combination of functional, structural, dynamic and static testing. Whatever the testing strategy is, whatever the mechanism used for error detection is, each correction leads inescapably to the risk of new errors appearance. It is therefore useful to make sure that the actions performed in order to correct the errors will not lead to new errors (software regression). This strategy is called *non−regression*

and is materialized by the re−execution, on the program tested, of a significant part of the old test data.

After this overview of software testing we relate, in the next session, our practical experience of software testing with PILOT interpreter.

5. TESTING THE INTERPRETER SOFTWARE

The Software Development Process

As mentioned above, some errors are inherent to the software development process itself. As far as the interpreter is concerned, its global behavior has been modeled by a finite state automaton [4], and interpretation algorithms have been defined for the various structures of the language PILOT [3]. The finite state automaton and the algorithms provide a good basis for avoiding some errors (errors due to information distortion, etc.). Nevertheless, no rigorous verification (formal or other) has been applied neither to the interpreter automaton nor to the interpretation algorithms. The control system software has been developed and modified by different persons with different programming experiences (less/more experienced).

The Testing Environment

Robots are reactive systems and the control system (and consequently the interpreter) in charge of the plan execution, has to deal with events generated by the robot. Such events are generally difficult to master and their presence increase the complexity of testing operations. Another important point is the impact of a test performed directly on the real robot. In fact, the consequences of a test are often unpredictable and the robot or its environment may be damaged. Our solution to these problems has been to build a simple simulator for the robot for our testing operations.

The Testing Strategy

For efficiency reasons, the interpreter testing has been performed by other persons than the initial developers. In fact, when the program is written and tested by the same person, the test is, most often, thoughtlessly less objective. Two main approaches have been used:

a− Static testing: this approach has been the first one applied to the interpreter software. It has consisted in reading the source code in order to detect programming errors such as those mentioned in the subsection "classification of errors" of section 4, and also in analyzing the source code in comparison with the interpretation algorithms and the semantics of PILOT available in [3].

b− Dynamic testing: test cases have been defined and applied to the binary code of the interpreter. The test data have been defined using first a functional approach and secondly the feedback from test shots.

In the two next subsections we detail these two testing steps.

Static Testing

The reading and analysis of the source code has led to the detection of the following errors:

a−Wrong interrupt handling: The code destined to the interpretation of a parallel structure contained the following sequence:

```
/* For the termination of the parallel box, one awaits the end
of the elementary actions launched in the parallel box. If there
are elementary actions still executing, one awaits their end as
follows */
For (index = 0; index < MAX_NB; index++)
{
    if ((elem_actions[index]→state == ON) && (...))
    {
        ...
        MessageHandling
        ...
    }
    ...
}
```

elem_actions is an array containing all the elementary actions launched since the beginning of the plan execution. According to the comments appearing in the program, the goal of the calls to *MessageHandling* in the for−loop is to wait for the end of the elementary actions of the parallel box which are still executing. Given that the interpreter may be interrupted at any time, the execution of the code above may be interrupted just before the call to *MessageHandling*, after a test on an action whose state is ON (an action under execution). The interrupt may itself originate from the end of the same action and its handler (which is also the *MessageHandling* procedure) will have been executed. Therefore, if this action is the last one under execution in the parallel box, the *MessageHandling* call of the for−loop will never end. As a consequence, a deadlock situation will occur in the interpretation of the plan. The same error appears in the interpretation of preemption. An error of the same nature also exists in the loop "**While** (*AlreadyProcessed* (Courant)) **loop**" of the procedure *Interpret.*

b−Wrong handling of continuous actions termination: The code appearing after the for−loop described above consists in stopping the continuous

actions of the parallel box. In fact, the following assumption is done: the for−loop ends when all the elementary actions have ended their execution. But this isn't true since the state of an elementary action may change from PRECONDITION_REQUEST to ON just after the evaluation of the for−loop condition. In such a case, the for−loop may end whereas an elementary action of the parallel box is still running. Failures due to this error have been observed during the use of the software (execution of the action following a parallel box whereas an action of the parallel box is still running, etc.).

c−Inexperienced programmer errors: Among these errors, we can cite the modification of the index of a for loop, functions returning a pointer to a local variable, etc.

The wrong interrupt handling and the wrong handling of continuous actions termination are conceptual errors whereas the other errors mentioned above are programming errors. These errors have been corrected before applying dynamic testing to our software. In the next subsection, we describe the dynamic testing of the interpreter software.

Dynamic Testing

a− Defining the representative test data: Our approach for the definition of the representative test data has been an *incremental approach*. We have started by testing an empty sequence, then the primitives of the language have been tested individually. Thereafter we have considered three combinations of the primitives of the language:

- *Combination in length* by increasing the number of elements in the sequences of the plan.
- *Combination in width* by increasing the number of branches within parallelism, preemption or conditional.
- *Combination in depth* by increasing the encapsulation level.

The following questions are raised. How to choose the proper length, width or depth? What are the pertinent combinations? Our choices have resulted from both the feedback from test shots and the following assumptions:

- Elementary actions are interchangeable and so are continuous actions.
- The set of sequences resulting from the combination of any two primitives of the language is representative of all the possible sequences of the language made of two or more primitives, except for memory space issues.
- The set of valid combinations of any two branches is representative of all the possible combinations in width of the language of two or more branches, except for memory space issues. The validity mentioned here is relative to the encapsulating primitive. For example, in the case of parallelism, one of the two branches should not contain continuous actions.
- The set of combinations in depth of encapsulation level 2 is representative of all the possible combinations in depth of the language whose encapsulation level is greater or equal to 2, except for memory space issues.
- The test of conditional iteration with a condition set to true, such as to execute an important number of loops, covers a wide range of memory space management errors.
- For a given encapsulating primitive, if n is its maximum number of branches ($n \geq 2$), then any combination in width of n branches is representative of all the combinations in width of 2 or more branches of that primitive, for memory issues.

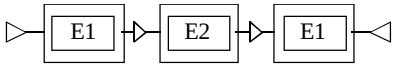

Fig. 1 A test with a sequence of actions

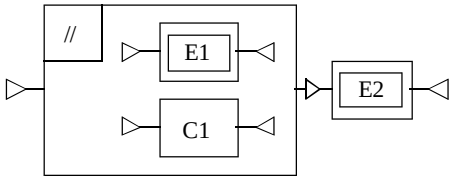

Fig. 2 A test with parallelism

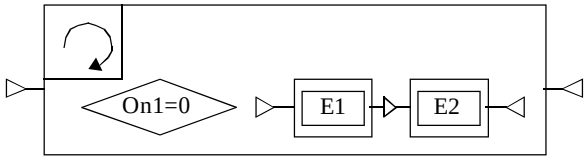

Fig. 3 A test with conditional iteration

b−Errors detected: Several test data (plans) have been defined following the approach described above. Figures 1 to 3 show some of the test data whose execution led to interpretation errors.

The execution of the plan of figure 1 revealed a problem in the interpretation of plans containing several occurrences of the same action (case of the action E1 in the plan). The second occurrence of this action was never executed. After analyzing the problem (debugging), we found the source of the problem: upon reception of the message PRECONDITION_ACCEPTED (this message includes the name of the action for which the

precondition is accepted), the interpreter did a search of the action into a list and launched the first one matching the name of the action. For the plan of figure 2, the action following the parallel box was executed twice. The analysis of the problem revealed that the cause was a redundant handling of parallel execution termination.

As far as the plan of figure 3 is concerned, it has led to 2 main errors detection. The first one was a general memory space management problem concerning all plans: after few loops a BUS ERROR exception was raised. This error was due to a wrong size for the memory space allocated for the execution of plans. The second error was specific to iteration: in fact, a new structure was allocated for the execution of each primitive of the plan and this led to memory space problems when an important number of loops was executed. This problem has been solved by allocating only the space needed for the execution of one loop and reusing it for the various loops.

6. TOWARDS A FORMAL APPROACH FOR INTERPRETER TESTING

The benefits of the application of code reading and dynamic testing to the interpreter software is immediate: very few bugs have been observed since this work. Nevertheless, it doesn't ensure the conformance of the interpretation algorithms to the semantics of PILOT. For this reason, we have modeled and simulated the execution of the interpretation algorithms of PILOT with colored Petri nets. The idea is, in a first step, to verify the validity of the existing algorithms with respect to the operational semantics of PILOT and, in a second step, to proceed to a revision of the interpretation model if needed, and finally to generate safe interpretation algorithms from the validated model. This work is ongoing.

7. CONCLUSION

In this paper, we have related our practical experience in software testing of PILOT interpreter. The application domain has an impact on the testing activity. In our case, the context is that of reactive systems (robots) and the events generated in such systems are often difficult to master. This adds some complexity to testing operations. Another difficulty is the potential risk of direct testing on the real system (the robot or its environment may be damaged). To avoid this problem, we have built a simple simulator. Only "all−or−nothing" sensors have been simulated.

Two testing approaches have been applied to the interpreter software: static testing consisting in reading and analyzing the source code, and dynamic testing. Each of them has enabled the detection of errors of different nature (conceptual errors in the handling of interrupts and in the management of continuous actions termination, programming errors, etc.). The errors detected have been corrected.

Our incremental approach for the choice of representative test data has been presented. The test data have been generated manually. Future work will include the automatic generation of test data set based on the rules defined in our approach, the implementation of an environment for the automatic testing of the control system. All our tests shots have been done interactively and each test is time consuming. The automation of the testing process will reduce considerably the testing time and will make it possible to run more tests and to cover more errors. A more realistic model of the simulator is also necessary in order to reproduce some categories of errors (errors whose activation depends on the frequency of messages or events, etc.).

8. REFERENCES

[1] B. Beizer. "Software Testing Techniques, 2nd Edition", Van Nostrand Reinhold, 1990.

[2] E. Le Rest, J.L. Fleureau and L. Marcé. "PILOT: semantics and implementation of a language for telerobotics", In IROS'97, IEEE, Grenoble, France, September 1997.

[3] E. Le Rest. "PILOT: un langage pour la télérobotique", PhD thesis, Université de Rennes 1, France, 1996.

[4] J.L Fleureau. "Vers une méthodologie d'un système de programmation de télérobotique: comparaison des approches PILOT et GRAFCET", PhD thesis, Université de Rennes 1, France, 1998.

[5] J.L Fleureau, L. Nana Tchamnda, L. Marcé and L. Abalain. "Remote−Controlled Vehicle Using PILOT language", In ANS'99, American Nuclear Society, Pittsburgh, Pennsylvania, 1999.

[6] L. Nana Tchamnda and L. Marcé. "Vers une programmation sûre avec PILOT", MSR'2001, Colloque Francophone sur la Modélisation des Systèmes Réactifs, Toulouse, France, October 2001.

[7] V.A Nicolas. "Preuves de propriétés de classes de programmes par dérivation systématique de jeux de tests", PhD thesis, Université de Rennes 1, France, 1998, pp. 9−52.

[8] S. Xanthakis, M. Maurice, A. De Amescua, O. Houri and L. Griffet. "Test & contrôle de logiciels: Méthodes, Techniques & Outils", EC2, 1994.